CoGrammar

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

Working with SQL in Python

Background

As software engineers and data scientists, you are undoubtably going to come across databases, they are important to most of the work that you will do in the professional space

Software Engineer

- Persisting application data
- Sharing of information

Data Scientist

- Access to business data
- Working with large teams
- More reliable than local files

Objectives

We will be focused on:

- Connecting to SQLite in Python
- See how to perform database operations
- Connect to another type of database
- Connect our database to Pandas
- Look at ORMs

SQLite

What is SQLite

- Lightweight database
- Does not require a server
- Good for storing local data
- Bad for large scale applications

Other Notes

- Comes preinstalled with Python
- Comes preinstalled on android devices

Working with SQLite in Python

1. Import the module

```
import sqlite3
```

2. Create a database connection

```
db = sqlite3.connection('my_database.db')
```

3. Create a cursor to execute statements

```
cursor = db.cursor()

cursor.execute("""
    CREATE TABLE student(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        grade INTEGER
    );
""")

result = cursor.fetchall()
print(result)
```

Working with SQLite: Execute Method

We can pass values to our statements using arguments in the excute method

```
name = "Andres"
grade = 60

cursor.execute("INSERT INTO student(name, grade) VALUES (?, ?)", (name, grade))

db.commit()
```

If we want to insert a single value, we still need the comma to show that we are passing a tuple and not a normal data type variable

```
name = 'John'

cursor.execute("""
    SELECT *
    FROM people
    WHERE name = ?
""", (name,))

conn.commit()
```

Working with SQLite: executemany Method

If we want to perform the same operation on a list of values, we can use the executemany method to pass each value to the statement

```
students = [
    ("Jack", 50),
    ("Tommy", 60),
    ("Jerry", 70),
    ("Bob", 80)
]

cursor.executemany("""
    INSERT INTO student (name, grade)
    VALUES (?, ?)
""", students)

conn.commit()
```

Working with SQLite: Retrieving data

We can use the fetch method after performing a SELECT statement to get the results of our query

```
id = 3

cursor.execute("""
    SELECT *
    FROM student
    WHERE id = ?
""", (id,))

print(cursor.fetchone())
```

Working with SQLite: Retrieving data

If we would like to get more than 1 record, we can use the fetchall() method

```
cursor.execute("""
    SELECT name, grade
    FROM student
""")

students = cursor.fetchall()

print(students)
```

Working with SQLite: Closing the connection

Once we have performed our operations, it's important to make sure that the connection is closed, the approach you take to closing your database connection depends on a few factors:

- How frequently will we be making requests to the database
- Is this the only database we are working with in the application
- How many files (modules) need access to the database

Approaches

- Opening and closing the connection after every operation
- Keeping the connection open for the life of the application

Exception Handling and Security

When working with anything that you do not have direct control over in your code, there is a chance that it might throw an error.

A few things that might cause an error in SQL

- Failed Connection
- Statements - eg, table doesn't exist, table already exists ...
- Too many database connections
- and more

Important Ones

- Failed connection
  - before connecting, it's good to have a try-except to make sure the database is available
- Too many database connections
  - Making sure that you close your databases will take care of this issue

Exception Handling and Security : Exception Handling

Handling an exception when connecting to a database

```
try:
    db = sqlite3.connect('database')
except:
    print('Error connecting to the database')
    db.close()
    exit()

cursor = db.cursor()
...
```

Exception Handling and Security : Handling Database Locked

To make sure that there is always a single connection, it's good to either pass a single connection to the functions that are working with the database, or open and close the database within each function.

If we want to get more advanced with OOP, we would have a single class that handles our database interactions and pass this object to the different places that need to access the database

```
def get_grade(student_id: int):
    conn = sqlite3.connection('student.db')

    cursor = conn.cursor()

    cursor.execute("""
        SELECT grade
        FROM student
        WHERE id = ?
    """, (student_id,))

    result = cursor.fetchone()

    conn.close()

    return result
```

Exception Handling and Security : Security

Since the execute method uses strings, you might be tempted to use f-strings to pass values, but there is a reason why the method takes in arguments the way it does. This is to protect against SQL Injections

What is SQL Injection

When a bad actor passes a SQL statement into an application in order to perform malicious operations

Execute Method

The execute method by default guards against SQL injection in 2 ways

1. Through the use of `?` when passing arguments, these arguments are cleaned before they are passed to the statement
2. Only allowing a single statement to be run at a time.

Connecting to Other Databases

SQLite is not the best option for most use-cases due to it's limited scope, there are many other SQL based options, the most popular are:

SQL Server

PostgreSQL

Connecting to Other Databases: Connection

Most other databases will require some sort of hosting, you have 3 main options, each with their own pros and cons, but it's up to you to make the choice

Local Install

Cloud

Docker

Connecting to Other Databases: Installing Locally

Pros

- Offline access to database
- Usually comes with a DBMS tool like MS SSMS for example

Cons

- Hard to set up
- Always running in the background
- Might leave you working with a single database on all projects

Connecting to Other Databases: Using Docker

Docker allows you to run tools and services in an isolated environment

Pros

- Set up and teardown databases as you need them
- Makes collaboration a lot easier, everyone can work on the same version of the database
- Allows you to work offline

Cons

- A bit of a learning curve
- Image sizes can be large

Connecting to Other Databases: Cloud Database

Pros

- Can be connected to from anywhere
- Really easy to set up

Cons

- Can be costly since database needs to be hosted
- Does not allow for offline work

Connecting to Other Databases: Connecting in Python

To connect to a database, we will first need to install the correct backage for our database, just search the package that works with your database

- SQL Server - `pyodbc`
- PostegreSQL - `psycopg`
- MySQL - `mysql-connector-python`

Working With the Database

Most of the tools we use will have the exact same methods as the sqlite3 module, so everything will be exactly the same once we've connected to the database

Connecting to Other Databases: Connecting to Database

Like SQLite, we will need a single string to connect to our database, the only difference is that this single string contains a bunch of information about our connection, always read the documentation for your packages.

Lets connect to a SQL Server instance

1. Install `pyodbc`
2. Create connection string
3. Connect to database
4. Have fun

Connecting to Other Databases: Connecting to SQL Server

```
pip install pyodbc
```

```
import pyodbc

connection_string = """
    Driver={SQL Server Native Client 11.0};Server=server_name;Database=DatabaseName;Trusted_Connection=yes;
"""
```

```
conn = pyodbc.connect(connection_string)
```

```
cursor = conn.cursor()
cursor.execute("""
    SELECT TOP(5) *
    FROM product
""")
```

LETS GET CODING!!